CS 635 Advanced Object-Oriented Design & Programming
Fall Semester, 2022

Doc 06 Null Object, Strategy
Sep 13, 2022

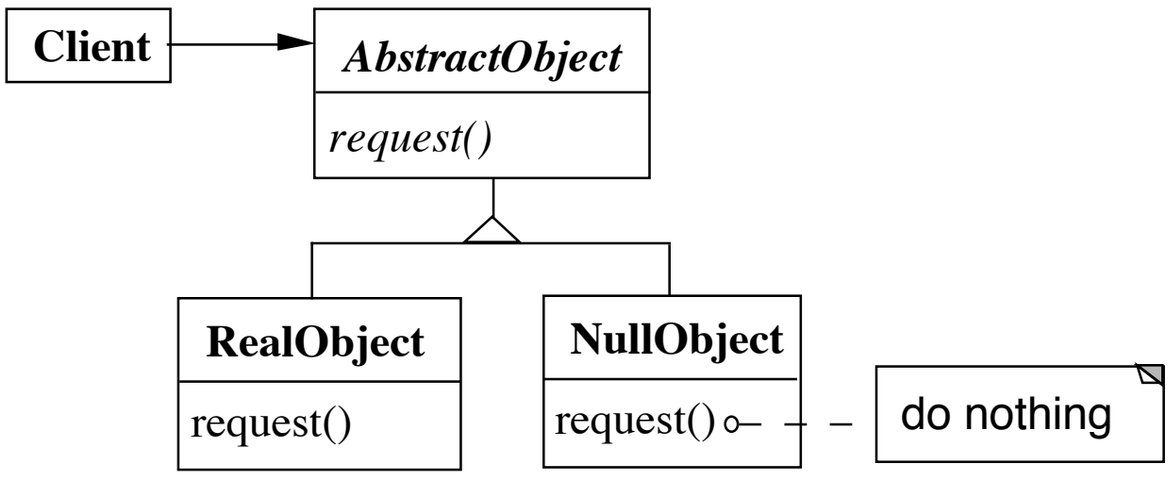
Why Software Projects Go Wrong

More software projects have gone awry for lack of quality, which is part of many destructive dynamics, than for all other causes combined.

Gerald M. Weinberg

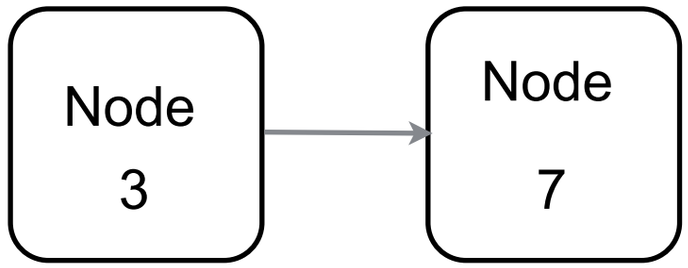
Null Object

Null Object



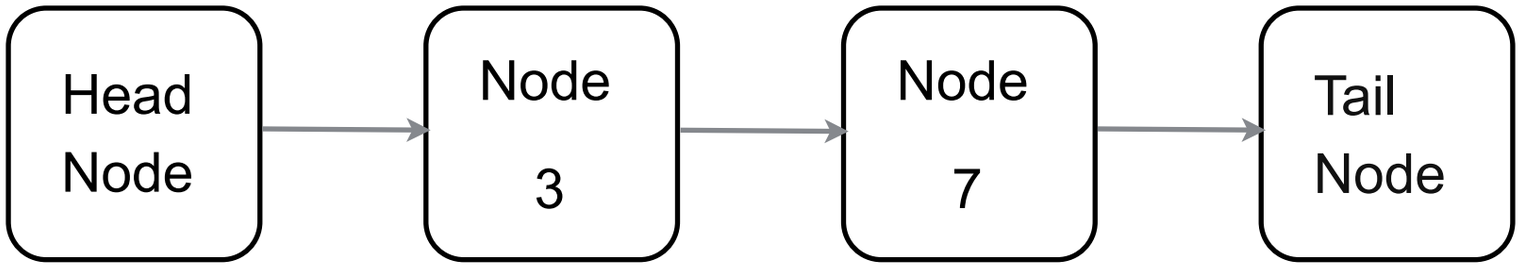
NullObject implements all the operations of the real object,

These operations do nothing or the correct thing for nothing




```
class LinkedList {
   Node head;

   public toString() {
      if (head == nil) {
         return “()”;
      }
      String listAsString = “(;
      Node current = head;
      while (current != null) {
         listAsString += current.value() + “, “;
         current = current.next;
      }
      listAsString = removetail(listsString, 2);
      return listAsString + “)”;
   }
```




```
class LinkedList {
   Node head;

   public toString() {
      return head.toString();
   }
```

```
class HeadNode {
   public String toString() {
      return "(" + next.toString();
   }
}
```

```
class Node {
   public String toString() {
      return " " + element + next.toString();
   }
}
```

```
class TailNode {
   public String toString() {
      return " )";
   }
}
```

Applicability - When to use Null Objects

Some collaborator instances should do nothing

You want clients to ignore the difference between a collaborator that does something and one that does nothing

Client does not have to explicitly check for null or some other special value

You want to be able to reuse the do-nothing behavior so that various clients that need this behavior will consistently work in the same way

Applicability -When not to use Null Objects

Very little code actually uses the variable directly

The code that does use the variable is well encapsulated

The code that uses the variable can easily decide how to handle the null case and will always handle it the same way

Consequences

Advantages

Uses polymorphic classes

Simplifies client code

Encapsulates do nothing behavior

Makes do nothing behavior reusable

Disadvantages

Forces encapsulation

Makes it difficult to distribute or mix into the behavior of several collaborating objects

May cause class explosion

Forces uniformity

Is non-mutable

Implementation

Too Many classes

Multiple Do-nothing meanings

- Try Adapter pattern

Transformation to RealObject

- Try Proxy pattern

Refactoring: Introduce Null Object

You have repeated checks for a null value

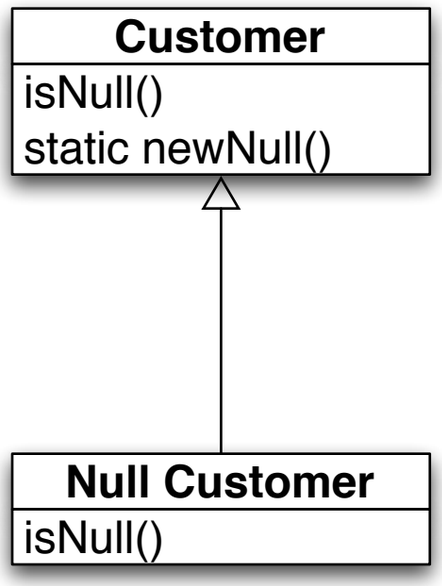
Replace the null value with a null object

```
if (customer == null)
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```

```
plan = customer.getPlan();
```

||

Create Null Subclass



public boolean isNull() { return false;}
public static Customer newNull() { return new NullCustomer();}

boolean isNull() { return true;}

Compile

Replace all nulls with null object

```
class SomeClassThatReturnCustomers {

   public Customer getCustomer() {
      if (_customer == null )
         return Customer.newNull();
      else
         return _customer;
      }
      etc.
}
```

Compile

Replace all null checks with isNull()

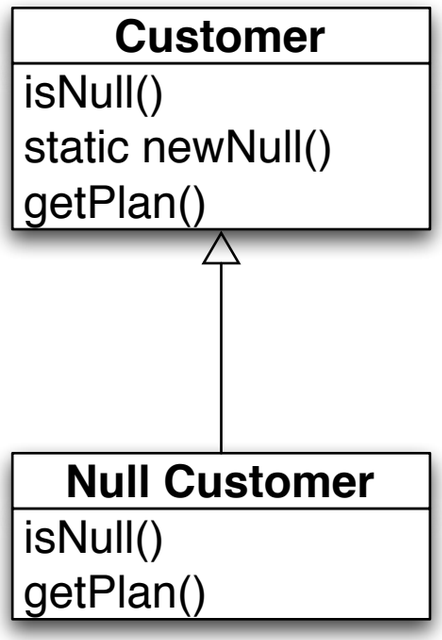
```
if (customer == null)
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```

```
if (customer.isNull())
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```

Compile and test

Find an operation clients invoke if not null

```
if (customer.isNull())
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```




```
class NullCustomer {
   public BillingPlan getPlan() {
      return BillingPlan.basic();
   }
```

Remove the Condition Check

```
if (customer.isNull())
   plan = BillingPlan.basic();
else
   plan = customer.getPlan();
```

```
plan = customer.getPlan();
```

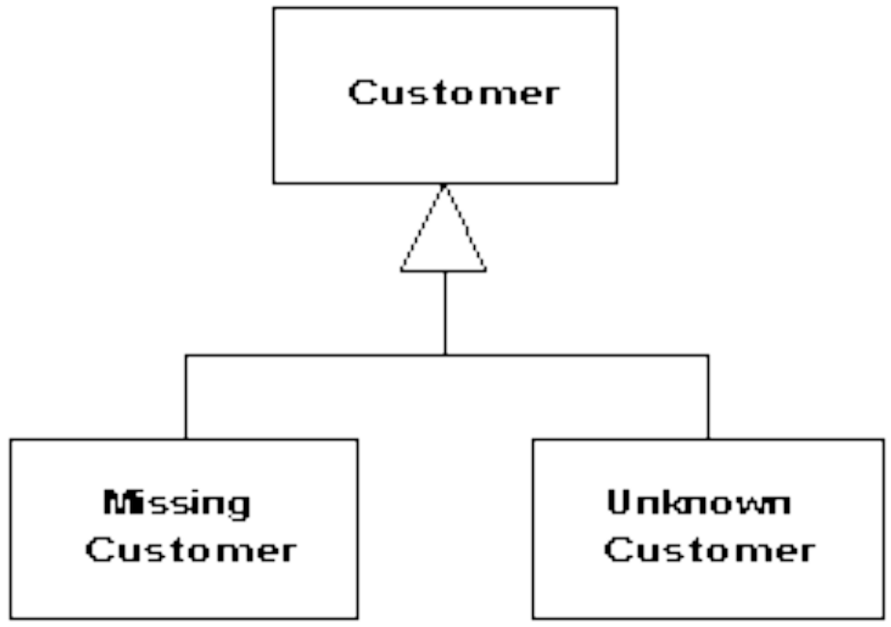
Compile & Test

Repeat last two slides for each operation
clients check if null

Special Case

Special Case

Represent special cases by a subclass



Use when multiple places that have same behavior

After conditional check for particular class instance

Or same behavior after a null check

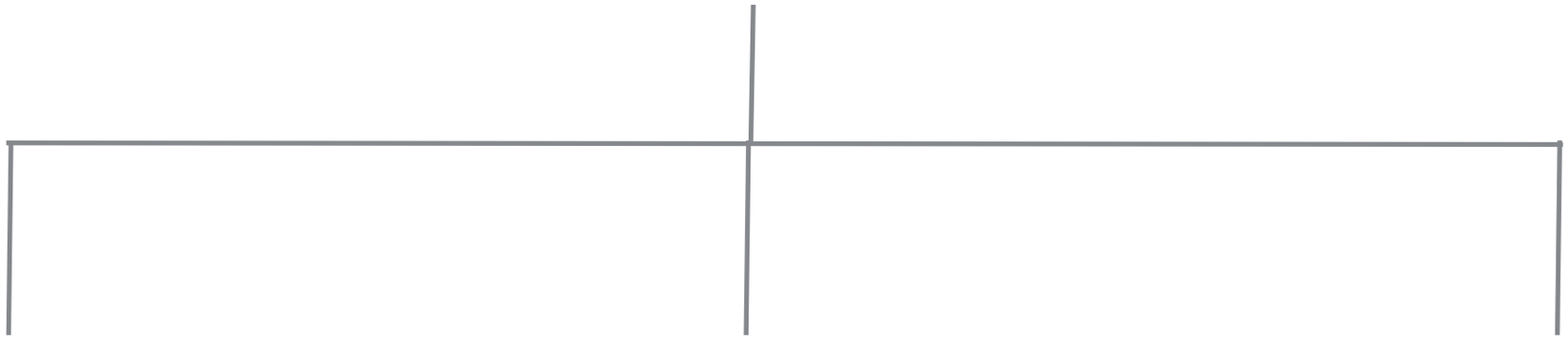
Strategy Pattern

Favor
Composition
over
Inheritance

Orderable List

Sorted
Reverse Sorted
Random

OrderableList



SortedList ReverseList RandomList

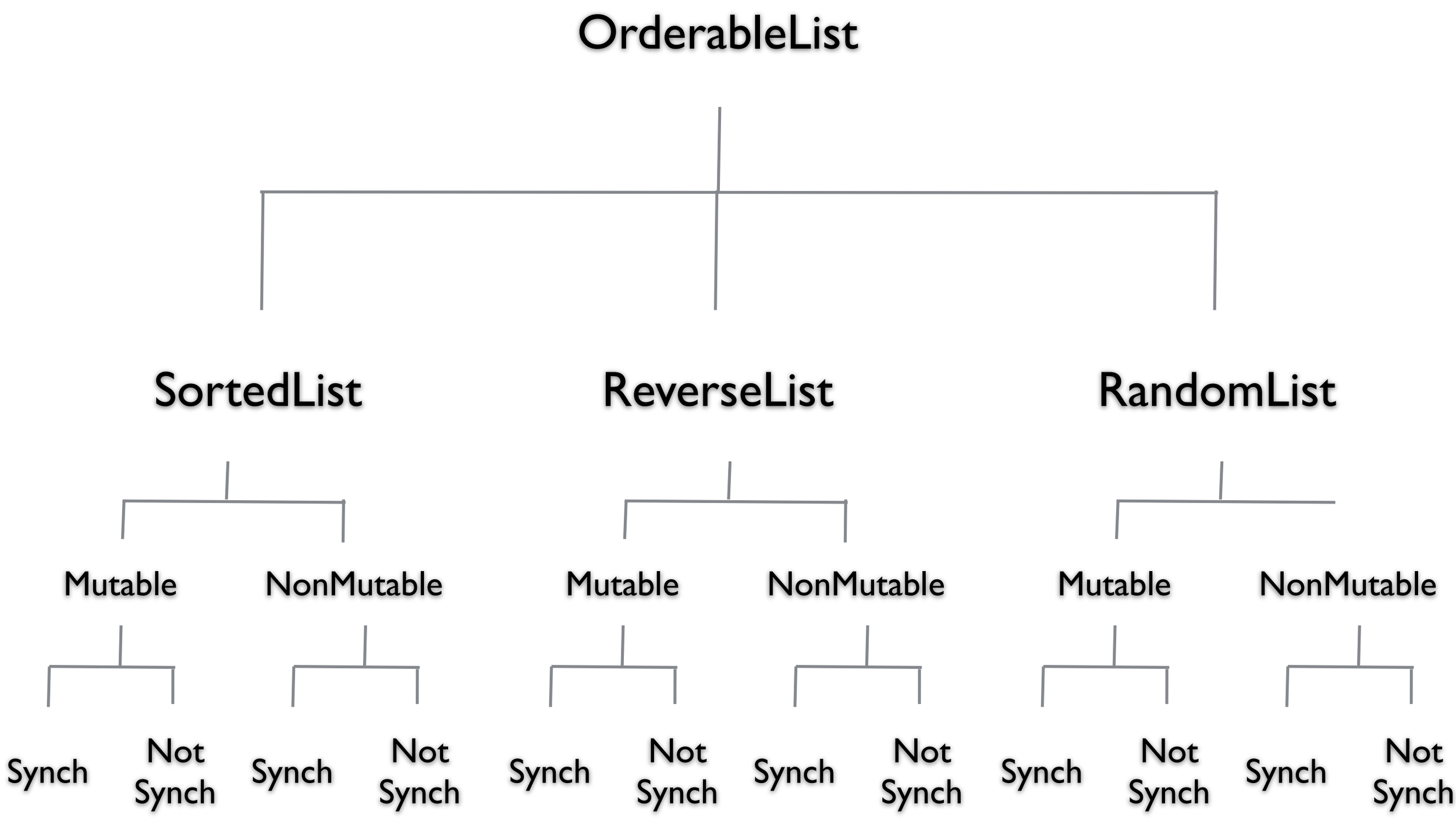
One size does not fit all

Issue 1 - Orthogonal Features

Order
Sorted
Reverse Sorted
Random

Threads
Synchronized
Unsynchronized

Mutability
Mutable
Non-mutable


OrderableList
SortedList
ReverseList
RandomList
Mutable
NonMutable
Mutable
NonMutable
Mutable
NonMutable
Synch
Not Synch
Synch
Not Synch
Synch
Not Synch
Synch
Not Synch
Synch
Not Synch
Synch
Not Synch

Issue 2 - Flexibility

Change behavior at runtime

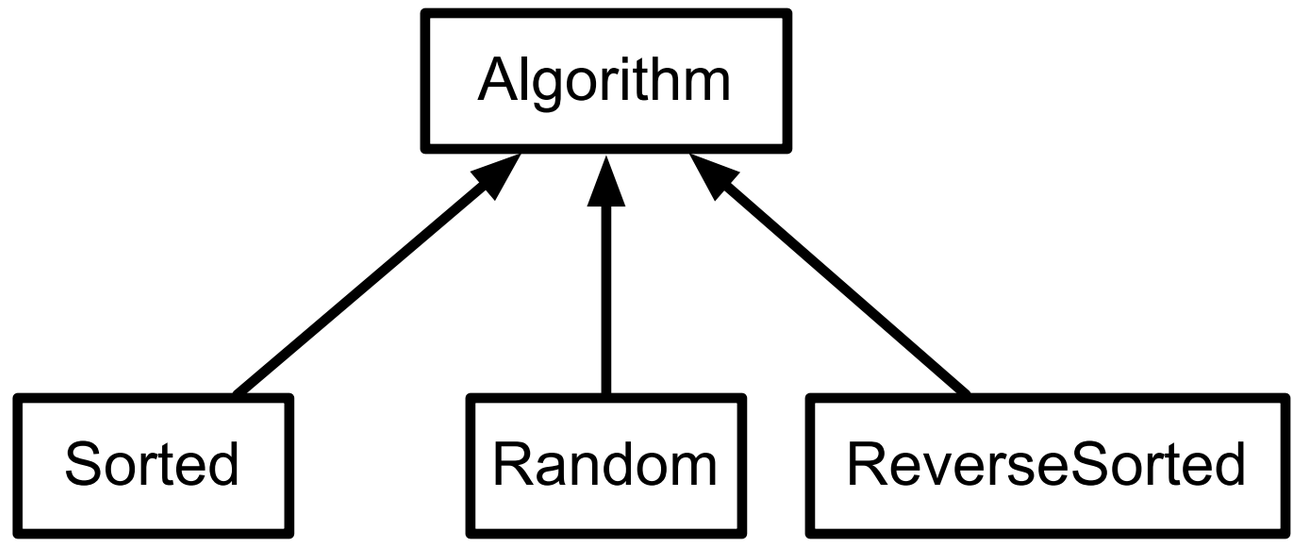
```
OrderableList x = new OrderableList();
x.makeSorted();
x.add(foo);
x.add(bar):
x.makeRandom();
```

Configure objects behavior at runtime

Strategy Pattern

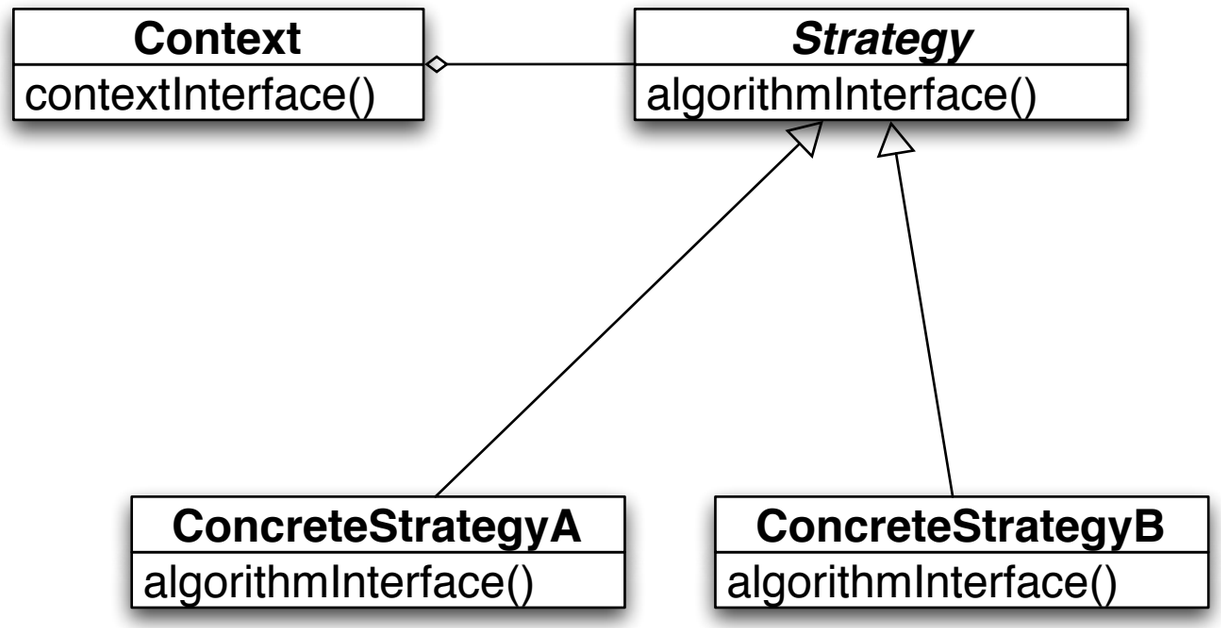
```
class OrderableList {
    private Object[ ] elements;
    private Algorithm orderer;

    public OrderableList(Algorithm x) {
       orderer = x;
    }
```




```
public void add(Object element) {
   elements = ordered.add(elements,element);
}
```

Structure


Context
contextInterface()
Strategy
algorithmInterface()
ConcreteStrategyA
algorithmInterface()
ConcreteStrategyB
algorithmInterface()

The algorithm is the operation

Context contains the data

How does this work?

Prime Directive

Data + Operations

How does Strategy Get the Data?

Pass needed data as parameters in strategy method

Give strategy object reference to context
Strategy extracts needed data from context

Example - Java Layout Manager

```
import java.awt.*;
class  FlowExample  extends Frame  {

    public FlowExample( int  width, int height ) {
        setTitle( "Flow Example" );
        setSize( width, height );
        setLayout( new FlowLayout( FlowLayout.LEFT) );

        for ( int label = 1; label < 10; label++ )
            add( new Button( String.valueOf( label ) ) );
        show();
    }

    public  static  void  main( String  args[] ) {
        new  FlowExample( 175, 100 );
        new  FlowExample( 175, 100 );
    }
}
```

Example - Smalltalk Sort blocks

```
| list |
list := #( 1 6 2 3 9 5 ) asSortedCollection.
Transcript
    print: list;
    cr.
list sortBlock: [:x :y | x > y].
Transcript
    print: list;
    cr;
    flush.
```

Java Sorting

How to sort a Collection in Java?

~~ArrayList~~ List method - sort(Comparator<? super E> c)

Create a subclass of Comparator

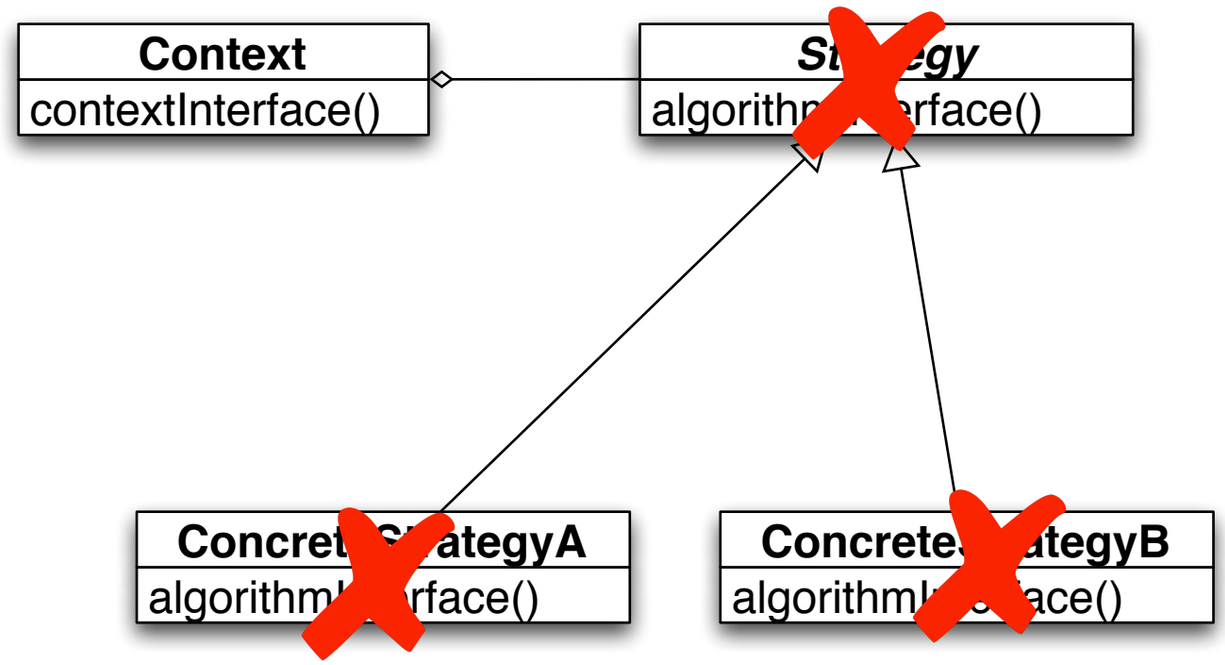
Pass in comparator object to sort method

```
List<Students> students = new ArrayList<>();
add students
students.sort(new SortByGPA());
```

Lambda & Strategy Pattern

If strategy only contains one method

Can replace Strategy classes with a lambda



In Java may need to define lambda type

Java Sorting Using Lambda

```
List<Students> students = new ArrayList<>();

add students

students.sort( (a, b) -> (a.gpa() <= b.gpa()) ? -1 : 1);
```

Costs

Clients must be aware of different Strategies

Communication overhead between Strategy and Context

Increase number of objects

Benefits

Alternative to subclassing of Context

Eliminates conditional statements

Replace in Context code like:

```
switch  ( flag ) {
    case A: doA(); break;
    case B: doB(); break;
    case C: doC(); break;
}
```

With code like:

```
strategy.do();
```

Gives a choice of implementations

Refactoring:

Conditional logic in a method controls which of several variants of a calculation are executed

so

Create a Strategy for each variant and make the method delegate the calculation to a Strategy instance

Replace Conditional Logic with Strategy

```
class Foo {
   public void bar() {
      switch  ( flag ) {
         case A: doA(); break;
         case B: doB(); break;
         case C: doC(); break;
      }
   }
}
```

```
class Foo {
   private strategy;
   public void bar() {
      strategy.do(data);
   }
}
```